2015 IEEE Symposium on Security and Privacy

Cross-Architecture Bug Search in Binary Executables

Jannik Pewny*, Behrad Garmany*, Robert Gawlik*, Christian Rossow*†, Thorsten Holz*
*Horst Görtz Institut (HGI) Ruhr-Universität Bochum
†Cluster of Excellence, MMCI Saarland University

***Abstract*—With the general availability of closed-source software for various CPU architectures, there is a need to identify security-critical vulnerabilities at the binary level to perform a vulnerability assessment. Unfortunately, existing bug finding methods fall short in that they i) require source code, ii) only work on a single architecture (typically x86), or iii) rely on dynamic analysis, which is inherently difficult for embedded devices.**

In this paper, we propose a system to derive *bug signatures* for known bugs. We then use these signatures to find bugs in binaries that have been deployed on different CPU architectures (e.g., x86 vs. MIPS). The variety of CPU architectures imposes many challenges, such as the incomparability of instruction set architectures between the CPU models. We solve this by first translating the binary code to an intermediate representation, resulting in *assignment formulas* with input and output variables. We then sample concrete inputs to observe the I/O behavior of basic blocks, which grasps their semantics. Finally, we use the I/O behavior to find code parts that behave similarly to the bug signature, effectively revealing code parts that contain the bug.

We have designed and implemented a tool for cross-architecture bug search in executables. Our prototype currently supports three instruction set architectures (x86, ARM, and MIPS) and can find vulnerabilities in buggy binary code for any of these architectures. We show that we can find *Heartbleed* vulnerabilities, regardless of the underlying software instruction set. Similarly, we apply our method to find backdoors in closed-source firmware images of MIPS- and ARM-based routers.

I. Introduction

Software bugs still constitute one of the largest security threats today. Critical software vulnerabilities such as memory corruptions remain prevalent in both open-source and closed-source software [40]. The National Vulnerability Databases listed 5,186 security-critical vulnerabilities in 2013, and trends from recent years suggest a steady number of software bugs. However, even though vulnerabilities are known, it is oftentimes challenging to tell which particular software is vulnerable, especially if software libraries are re-used by larger software projects [15]. Worse, state-of-the-art tools have deficits in handling libraries that become part of software that has been deployed on a variety of architectures.

The problem of finding bugs at the source code level has been addressed by a lot of researchers [14], [17], [18], [22], [42]. Professional code verification tools ensure source code quality and a number of automated bug finding proposals analyze source code to find security-critical bugs. However, access to source code is quite a rigid assumption when it comes to finding bugs. A lot of prominent software is available only as a binary, either as commercial software (e.g., `MS Office`) or as freely-available closed-source software (e.g., `Adobe Reader` or `Flash`). Software on embedded devices, typically referred to as *firmware*, is usually closed-source, implemented in an unsafe language, and re-uses (potentially vulnerable) code from third-party projects [8], [37]. What is more, closed-source software may be even stripped, i.e., the binaries do not contain symbol information like function names or data types. Thus, we seek a solution to find vulnerabilities at the binary level without requiring symbols. This alone makes the process to find bugs significantly more challenging than source code-level approaches.

Another challenge in finding bugs at the binary level is that more and more software is cross-compiled for various CPU architectures. That is, even if the bug is known for one particular architecture (say Intel x86), finding a bug stemming from the same source code, but used in a project for another architecture (say ARM), poses various difficulties: Binaries from varying architectures differ in instruction sets, function offsets and function calling conventions, to name but a few significant complications. This is problematic for many kinds of cross-compiled software. For example, hardware vendors use the same code base to compile firmware for different devices (e.g., home routers, cameras, VoIP phones) that operate on varying CPU architectures. Similarly, prominent software such as `MS Office`, `Adobe Reader` or `Flash`, is already available for multiple platforms and architectures, most recently with the increase of ARM-based `Windows RT` deployments. The problem is compounded if cross-compiled software includes well-known, but vulnerable libraries. For instance, after discovery of the *Heartbleed* bug in `OpenSSL`, there is a growing list of affected closed-source software running on various architectures (x86, MIPS, ARM, PowerPC, etc.). Currently, though, there is no practical way to automatically find such known bugs in binaries belonging to different architectures. Users have to resort to manual inspection or have to rely on the vendors to inspect the closed-source code for bugs. However, as the *Heartbleed* case illustrated once more, vendors are not particularly quick to evaluate security risks in their products. Oftentimes this leaves open a significant time window during which security-critical bugs can be exploited. Worse, there is no guarantee that end-of-life hardware or firmware from vendors that have disappeared over time is ever inspected for well-known bugs, although such systems are still widely used by both consumers and industry.

In this paper, we address this problem and make the first

DOI 10.1109/SP.2015.49

step towards finding vulnerabilities *in binary software* and *across multiple architectures*. As the cross-architecture feature complicates the bug search quite a bit, we focus on the following use case: Once a bug is known—in any binary software compiled to a supported architecture—we aim to identify equally vulnerable parts in other binaries, which were possibly compiled for other architectures. That is, we use a *bug signature* that spans a vulnerable function (or parts of it) to find similar bug instances. While this limits our work to re-finding similar, previously documented instances of bug classes, exactly this problem has evolved to a daily use case in the era of cross-compiled code. For example, *Heartbleed* affected a tremendous number of closed-source software products from multiple vendors across all CPU architectures. The list of affected software is still growing and there is no automated way to identify vulnerable software versions. Similarly, it was discovered that several Linksys and Netgear devices included a backdoor in their firmware, but users had to trust the vendors to name all affected products. Our goal is to provide a mechanism to assist a human analyst in such scenarios, where the analyst defines a bug signature *once* and then searches for parts in other software binaries—from and to any architecture—that contain a similar bug.

To this end, we propose a mechanism that uses a unified representation of binaries so that we can compare binaries across different instruction sets (i.e., across architectures). That is, we first lift binary code for any architecture—we currently support Intel x86, ARM and MIPS due to their high popularity—to an intermediate representation (IR). Even in this IR, we are bound to binary information lacking symbols and data types. Based on this IR code, we thus aim to grasp the *semantics* of the binary at a basic block level. In particular, we build assignment formulas for each basic block, which capture the basic block's behavior in terms of input and output variables. An input variable is any input that influences the output variables, such as CPU registers or memory content. We then sample random input variables to monitor their effects on the output variables. This analysis results in a list of input/output (I/O) pairs per assignment formula, which capture the actual semantics of a basic block. Although the *syntax* of similar code is quite different for the various CPU architectures (even in the intermediate representation), we can use such *semantics* to compare basic blocks across ISAs.

We use the semantic representation to find the bug signature in other arbitrary software that is potentially vulnerable to the bug defined in the signature. The bug signature can be derived automatically from a known vulnerable binary program or from source code, and may simply represent the entire vulnerable function. To preserve performance, we use *MinHash* to significantly reduce the number of comparisons between I/O pairs to find suitable basic block matches. Lastly, once basic block matches have been found, we propose an algorithm that leverages the control flow graph (CFG) to expand our search to the entire bug signature. As output, our system lists functions ordered by their similarity to the bug signature. This gives analysts a compact overview of potentially vulnerable functions in the analyzed binary.

To evaluate our approach, we first systematically test how well our system performs when matching equivalent functions across binaries that have been compiled for different architectures, with different compilers, and using varying optimization levels. The evaluation shows that our system is accurate in matching functions with only a few false positives. For example, we show that our approach ranks 61% of the `OpenSSL` ARM functions in a MIPS-based `OpenSSL` binary among the 10 closest function matches. Second, we evaluate our system in various real-world use cases, for which we extracted the vendor-compiled software binaries from firmware in order to search for real-world vulnerabilities. Our system finds the *Heartbleed* bug in 21 out of 24 tested combinations of software programs across the three supported architectures. Further, we find vulnerable `RouterOS` firmware and confirm backdoors in Netgear devices. Note that during all of these real-world tests we did not have access to the source code and thus used the actual software binaries contained in the corresponding firmware—highlighting that our design even tolerates deviations in build environments.

We obtained these results under certain assumptions, e.g., that the binaries have not been obfuscated. However, we also show that our system can tolerate (more common) binary disturbances to some extent, such as compiler optimizations or differences in build environments—in addition to tackling most of the previously-unsolved discrepancies in comparing code between ISAs from various architectures.

Furthermore, our method can be used with sub-function granularity, which is vital for bug search. While function-wise matching techniques (like BLEX [11], BinDiff [10] or Exposé [32]) could find bugs in functions that have been cloned, it is often useful to find re-used *parts of* a vulnerable function in other functions, instead of finding mere clones.

The uses of our proposed system are manifold; in this work we focus on identifying unpatched bug duplicates. However, we are not limited to doing so in the same program (i. e., the same binary), and do not even require that the target binary is compiled for the same architecture. For example, companies that run closed-source software from vendors that do not support particular software/devices anymore (e.g., if the product is beyond end-of-life or if the vendor goes out of business) could independently verify, whether common bugs are present. Similarly, if bugs in widely used libraries become known (such as *Heartbleed*, recently), CERTs can find affected products in a short amount of time. Our concept can also be applied to search for known backdoors in closed-source applications. We envision more use cases of our system, such as binary diffing, searching for software copyright infringement in binaries, or revealing code sharing across binaries.

In summary, our four main contributions are as follows:

- We lift ARM, x86 and MIPS code to unified RISC-like expressions that capture I/O syntax per basic block.
- We introduce a sampling and MinHashing engine to create compact and cheaply-comparable semantic summaries of basic blocks—the basis of our bug search.

- We define a metric to compare code structures like sub-CFGs and functions, which enables us to search for bug signatures in arbitrary software binaries.
- We empirically demonstrate the viability of our approach for multiple real-world vulnerabilities spanning software across three supported architectures.

II. Approach

We now outline our general approach for cross-architecture bug search in binary executables. Our goal is to use a code similarity metric to find code locations that are similar to code containing a bug. The assumption here is that similar code often stems from slightly modified shared code, which typically also shares the same bug. We are particularly interested in finding bugs that are security-critical. Our method, though, supports finding many types of bugs, and we thus use the words *bug* and *vulnerability* interchangeably.

A. Workflow

We use a *bug signature*, i.e., a piece of binary code that resembles a specific instance of a vulnerability class, to find possible vulnerabilities in another binary program (*target program*). To this end, we first derive a bug signature (Section II-B). Then, we transform both the bug signature and the target program into an intermediate representation (Section II-C) and build compact basic block-wise semantic hashes (Section II-D). All these transformations are a one-time process for both the bug signature and the target program.

Figure 1 illustrates this process for the instructions `ldrb` (ARM), `lbu` (MIPS) and `lodsb` (x86). First, we convert these assembly instructions to an intermediate representation, which results in a list of *assignment formulas* that we represent as easy-to-parse S-Expressions (symbolic expressions). The assignment formulas detail how an output variable is influenced by its inputs. For example, the first line in the x86 case represents that the first eight bits of the address where `ESI` points to are stored in register `AL`. The number of inputs differs for each formula (e.g., no input for the terminator, one input for the `AL`, `v0` and `R3` variables, or two inputs for the `ESI` variable). Next, using random concrete input values (dashed box), we sample the input/output behavior of these formulas (we illustrate sampling of the first formula only). For example, in the x86 formula of `ESI`, the concrete inputs $(5, 1)$ result in an output of 6. In the last step, we build semantic hashes over the I/O pairs, which allow us to efficiently compare the I/O behavior of basic blocks.

In the search phase, we use the transformed bug signature (i.e., its representation as a graph of assignment formulas) to identify bugs in the similarly-transformed binaries. That is, we look for promising matching candidates for all individual basic blocks of the bug signature in the target program (Section II-E). For each such candidate pair, we apply a CFG-driven, greedy, but locally-optimal broadening algorithm. The algorithm expands the initial match with additional basic blocks from the bug signature as well as the target program (Section II-F). The algorithm then computes the similarity between bug signature and target programs, returning a list of code locations ordered by their similarity to the signature. In the following, we explain these steps in more detail.

B. Bug Signatures

A bug signature is just like normal binary code: It consists of basic blocks and possible control-flow transitions between these basic blocks. Therefore, any selection of basic blocks can, in principle, be used as a bug signature. For example, the bug signature could represent an entire buggy function, limiting the manual effort to define the bug signature in more detail. However, users of our system should refine the bug signatures to smaller code parts, which cover only the bug and its relevant context. Note that our approach only requires a discriminative piece of code—typically, the context in which a bug occurs is so distinctive that our approach is completely independent from the vulnerability type. We have successfully evaluated buffer overflows, logical bugs and software backdoors.

However, it is quite hard to estimate how the signature size influences results in general: An additional non-characteristic basic block, which is essential to the structure of the vulnerability and discriminative in its context, will likely improve results, while a characteristic basic block, which is non-essential to the vulnerability, may lead to false positives.

While our target is to search bugs in binaries (i.e., without access to source code), we do not necessarily have to limit ourselves to use binary information only when deriving the bug signature. For example, consider the typical scenario that a bug in open-source software is publicized. This usually means that the buggy lines of code are known, which we can leverage to define the bug signature. Thus, we can use debug information to automatically find the basic blocks that correspond to the vulnerable function part, effectively deriving bug signatures with almost no manual effort.

Note that even if the buggy source code is available, source code-based bug finding techniques still cannot be applied if the vulnerable code became part of closed-source applications. In practice, buggy open-source code is re-used for closed-source applications, for which only the binary representation is available. For instance, vulnerabilities in the open-source software projects `BusyBox` and `OpenSSL` became part of many proprietary and closed-source firmware images.

C. Unifying Cross-Architecture Instruction Sets

Obviously, the instruction sets of architectures like x86, ARM, and MIPS are quite distinct. Aside from the instruction set, the calling conventions, the set of general- and special-purpose CPU registers, and the memory access strategies (e.g., load/store on RISC as opposed to CISC architectures like x86) also vary for each architecture. Even if binaries stem from the same source code, the resulting binary output cannot be compared easily if the binaries were compiled for different architectures. To illustrate this, Figure 2 shows a small code snippet that has been compiled for three architectures. It can be seen that not only the calling conventions and memory

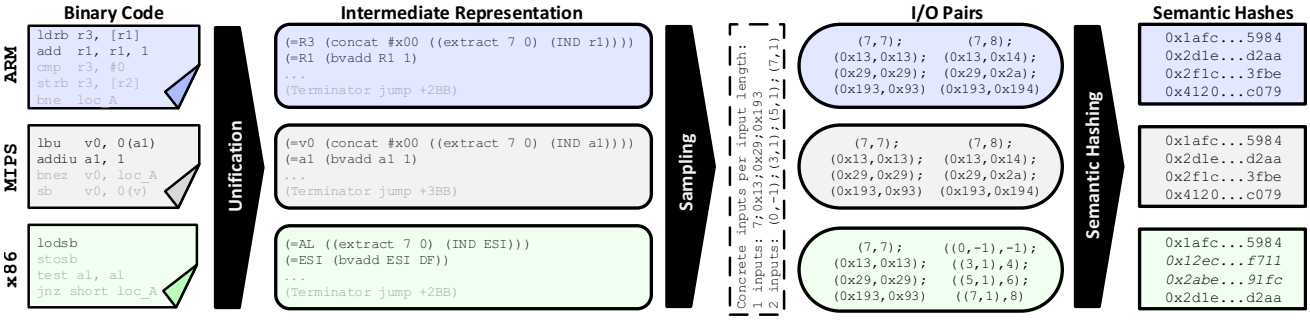


Fig. 1: Transformation phase: Three binaries (ARM, MIPS, x86) are first unified from assembly to the intermediate representation (S-Expressions), as illustrated for the first instruction (`ldrb`, `lbu` and `lodsb`). Then, using the same random concrete inputs among all formulas, we determine the input/output behavior of the expressions, resulting in a set of I/O pairs. In the last step, we create semantic hashes over these I/O pairs in order to efficiently compare basic blocks.

```
static void get_localtime(struct tm *ptm) {
  time_t timer;
  time(&timer);
  localtime_r(&timer, ptm); }
```

x86 assembly

```
timer= dword ptr -0Ch
ptm = eax    ; tm *
push    esi
mov     esi, ptm
push    ebx
sub     esp, 20h
lea     ebx, [esp+28h+timer]
push    ebx
call    time
ptm = esi    ; tm *
pop     ecx
pop     eax
push    ptm
push    ebx
call    localtime_r
add     esp, 24h
pop     ebx
pop     ptm
retn
```

ARM assembly

```
timer= -0x14
ptm = R0     ; tm *
STMFD   SP!, {R4,R5,LR}
SUB     SP, SP, #0xC
ADD     R4, SP, #0x18+timer
MOV     R5, ptm
MOV     ptm, R4
ptm = R5     ; tm *
BL      time
MOV     R0, R4
MOV     R1, ptm
BL      localtime_r
ADD     SP, SP, #0xC
LDMFD   SP!, {R4,ptm,LR}
BX      LR
```

MIPS assembly

```
li      $gp, 0xFFF9C
addu    $gp, $t9
addiu   $sp, -0x30
sw      $ra, 0x30+var_8($sp)
sw      $s1, 0x30+var_C($sp)
sw      $s0, 0x30+var_10($sp)
sw      $gp, 0x30+var_20($sp)
la      $t9, time
addiu   $s0, $sp, 0x30+var_18
move    $s1, $a0
jalr    $t9 ; time
move    $a0, $s0
lw      $gp, 0x30+var_20($sp)
move    $a0, $s0
la      $t9, localtime_r
nop
jalr    $t9 ; localtime_r
move    $a1, $s1
lw      $gp, 0x30+var_20($sp)
lw      $ra, 0x30+var_8($sp)
lw      $s1, 0x30+var_C($sp)
lw      $s0, 0x30+var_10($sp)
jr      $ra
addiu   $sp, 0x30
```

Fig. 2: `BusyBox` v1.21.1: C source of `get_localtime` and the corresponding `IDA` disassemblies.

accesses (e.g., load/store), but also the general assembly complexity/length, deviate between the architectures. Thus, at first, it seems hard to compare binary code across architectures.

To bridge this gap, a key step in our approach is to unify the binary code of different architectures. To this end, we first utilize an off-the-shelf disassembler to extract the structure of the binary code (such as functions, basic blocks, and control flow graphs). We then transform the complex instructions into simple, RISC-like and unified instructions. We do so for two reasons: First, it abstracts from architecture-specific artifacts and facilitates symbolic normalization. Second, later stages can now utilize this architecture-independent instruction set and therefore only have to be implemented once.

D. Extracting Semantics via Sampling

The unified instruction set would allow us to compare individual binary instructions syntactically. This already is a big advantage over comparing different instruction sets from multiple architectures. However, using the IR only solves some of the issues when comparing binary code from different architectures. It is also common to observe differences in calling conventions, register uses or memory accesses, which even influence the syntax of a unified representation. However, the *semantics* of the binary code will remain similar, even if source code is compiled for different architectures.

Therefore, in the next step, we aim to extract the semantics of the binary code. We aggregate the computational steps for each output variable of a basic block, which gives us precise *assignment formulas* for each output register or output memory location (see Figure 1, second column).

Special care needs to be taken for control flow transfers. By definition, every basic block ends with a terminator instruction, which determines the next basic block in the control flow. These terminators can have one successor (unconditional jumps, returns) or two successors (conditional jumps). To

abstract from concrete addresses, we define successors via symbolic function names (which, in our evaluation, are only available for imported functions, such as from `libc`), or via the number of basic blocks to skip. For example, the `true` case of a conditional jump may jump three basic blocks ahead. However, neither successors nor branch conditions are structurally different from other assignment formulas and are therefore handled in the same way. Note that indirect jumps, like returns, indeed have only one successor formula, even though this formula can be evaluated to multiple targets.

We extract the formulas per basic block, as we have observed that the control flow graph and the basic block margins typically remain equivalent (or at least similar) in cross-compiled code. Note that this assumption may not always hold for heavily obfuscated code or if compilers use varying code optimization strategies. However, as we will show in Section IV-B, our metric works quite well in scenarios with different architectures, different compilers, or different optimization levels. Section IV-C further shows that the metric can even find bugs in firmware of real-life, commercial devices.

For normalization purposes and simpler computation in later steps, we simplify the assignment formulas by passing them through a theorem prover. We therefore aggregate the RISC-like instructions and map them to the theorem prover's structure. The theorem prover then returns S-Expressions, i.e., an easy-to-parse data structure that represents the formulas and allows for arbitrary computations.

At this point, we have precise descriptions of the effect of a basic block on the programs state: The assignment formulas show an abstract representation of what operations the basic block will perform given symbolic input values (see Figure 1). However, these descriptions are still purely syntactic: Showing that two basic blocks are equivalent based on these formulas is—if possible at all—computationally intensive.

To achieve our goal of bug finding, we relax the condition to find *code equivalence* and use a metric that measures *code similarity*. Ideally, such a similarity metric gradually scales from 0.0 (distinct code) to 1.0 (equivalent code). We build such a metric upon *sampling*, which was proposed by Jin et. al [19]. First, we generate random and concrete values, which we use as inputs for the formulas of each basic block. Then, we observe the outputs of the formulas. Such point-wise evaluations capture the semantics of the performed computation. Arguably, the semantics are not captured perfectly, as not all possible values are used as input, but it is still reasonable to assume that similar computations have more input/output pairs in common than dissimilar ones.

In order to grasp the semantics of the formulas equally well, we have to devise a smart sampling strategy. Intuitively, the more input variables a formula has, the larger the number of sampled input variables should be. For example, the I/O-behavior of $a := b + 1$ can be grasped with fewer samples than the interplay of variables in $a := b * c + d$. Also, we have to make the samples introduce some robustness to the order of variables to make sure that, e. g., $a := b - c$ and $a := c - b$ can be recognized as similar. We show in Section III-B how we solved this issue by using permutations of the inputs.

Similarly, some formulas are prone to be misrepresented by the I/O pairs. For example, $a := b == 0$ will be `false` for all inputs but 0, such that it is semantically mostly similar to $a := 2 * b + 3 == 5$ (`false` for all inputs but 1). In such cases, we would have to find special "magic values" with a theorem prover, which is computationally expensive. Luckily, these cases usually occur only in branch conditions, which is why we chose to ignore formulas of branch conditions so that our similarity score is not biased.

E. Similarity Metric via Semantic Hashes

Evaluating a large number of sampled inputs results in many I/O pairs which represent the semantics of the evaluated basic blocks. The higher the number of shared, equal I/O pairs between two basic blocks, the higher is the similarity of these basic blocks. Thus, one could directly compare the I/O pair-sets of two basic blocks, e. g. with the Jaccard index, to measure their similarity. However, the large number of I/O pairs and basic blocks would cause such a naïve approach to scale badly, as it requires many I/O pair comparisons.

We tackle this bottleneck with locally-sensitive hashes. A LSH has the property that the similarity of the hash reflects the similarity of the hashed object. We chose to use MinHash [5], which satisfies the LSH properties and at the same time converges against the Jaccard index. In essence, MinHash computes a semantic hash over a basic block by incorporating the I/O pairs. Comparing two basic blocks now only requires comparing their semantic hashes instead of comparing the sets of I/O pairs, which significantly reduces the required complexity to measure the similarity of two basic blocks.

To compensate for statistical over-representation of the I/O pairs of multi-variable formulas and the property of the MinHash to operate on sets rather than multi-sets, we made two improvements to the straightforward application of the MinHashing algorithm (see Section III-C for details).

F. Comparing Larger Binary Structures

We have described how we capture the semantic representation of a basic block and presented a computationally cheap metric to compare two basic blocks. This metric allows us to perform the first step of comparing code: finding pairs of similar basic blocks, which are candidates for a signature-spanning match. In order to match an entire bug signature, though, comparing individual basic blocks is not sufficient. The code structure is quite relevant for some bug classes, e.g., for integer or buffer overflows, in which bound checks are either implemented or not. Bug signatures thus typically consist of multiple basic blocks and capture the structure of the code in terms of a CFG.

We therefore expand the comparison of individual basic blocks with an algorithm that aims to identify the entire bug signature. First, we pick a basic block from the bug signature and compare it to all basic blocks from the program in question. Then, we use an algorithm called *Best Hit Broadening* (BHB), which broadens the initial candidate match along the

```
// lodsb //
t0 = GET:I32(56)
t1 = GET:I32(32)
t8 = LDle:I8(t1)
PUT(8) = t8
t9 = Add32(t1,t0)
PUT(32) = t9
PUT(68) = 0x80579BE:I32
```

```
// ldrb r3, [r1] //
t11 = GET:I32(12)
t10 = Add32(t11,0x0:I32)
t13 = LDle:I8(t10)
t12 = 8Uto32(t13)
PUT(20) = t12
PUT(68) = 0xD816C:I32
```

```
// lbu v0, 0(a1) //
t4 = GET:I32(20)
t3 = Add32(t4,0x0:I32)
t6 = LDle:I8(t3)
t5 = 8Uto32(t6)
PUT(8) = t5
PUT(128) = 0x41ADA0:I32
```

Fig. 3: From left to right: VEX-IR of a load byte instruction on x86-32, ARM and MIPS.

Architecture	S-Expressions
ARM	(= R3 (concat #x000000 ((extract 7 0) (Indirection R1)))) (= R1 (bvadd R1 1)) (= PC #x000d816c)
MIPS	(= v0 (concat #x000000 ((extract 7 0) (Indirection a1)))) (= a1 (bvadd a1 1)) (= PC #x0041ada0)
x86	(= AL ((extract 7 0) (Indirection ESI))) (= ESI (bvadd ESI DF)) (= EIP #x080579be)

Fig. 4: S-Expressions

CFGs in the bug signature and target program. BHB operates in a greedy, but locally-optimal manner, until the match spans the entire signature. BHB is then repeated for all basic blocks in the bug signature, resulting in a list of functions ordered by their similarity to the signature (see Section III-D for details).

III. Implementation

In this section, we discuss some of the specific details for the proof-of-concept implementation of our approach.

A. Common Ground: The IR Stage

Our first design choice was which processor architectures we wanted to support. We chose ARM, x86 and MIPS (little- and big endian), because these architectures are pervasive and run many closed-source applications. x86 and ARM are nowadays the most widespread architectures. We additionally chose MIPS, as it is popular for embedded devices and is one of the few architectures that stay close to the RISC principles. We decided against x86-64, as it uses a different register size, which—without further adaptations of our approach—would inevitably lead to mismatches. In principle, other architectures can be supported and incompatibilities can be solved with some engineering effort.

First, we use `IDA Pro` [16] to extract a disassembly and the control-flow graph from the binary. Arguably, the resulting disassembly is not perfect [3], but it proved to be sufficient for our purposes.

Our next step was finding a common representation for binary code, which required us to consider the peculiarities of each architecture. For this purpose we utilize the VEX-IR, which is the RISC-like intermediate representation for the popular `Valgrind` toolkit. One of the benefits of VEX is its support for many architectures. VEX translates from binary to IR, but was never designed for static analysis, as it is part of a dynamic binary instrumentation framework. We refrain from discussing the VEX-IR in detail and refer the reader to the `Valgrind` documentation [39].

We leveraged `pyvex` [36], a Python framework with bindings to the VEX-IR, and used its API to process the IR statically. We feed binary opcodes to `pyvex` and dismantle the VEX statements into our own data structures. These data structures are used to aggregate and map the VEX-IR into semantically equivalent expressions for the `Z3` theorem prover [27]. The theorem prover's sole purpose is to simplify and normalize expressions. Additionally, it conveniently returns S-Expressions.

Figure 3 shows the first instruction of the second basic block of `BusyBox`' `strcpy` on each architecture. The load byte instruction operates implicitly (x86) or explicitly (ARM, MIPS) on registers. As all semantics of an instruction are made explicit and transformed to RISC operations, they serve as a convenient ground to deduce their effects and incorporate them into S-Expressions. The numbers in *GET* and *PUT* are offsets in a shadow table that are mapped to specific registers. In the x86 example in Figure 3, `t1 = GET:I32(32)` loads the content of register `ESI` into the temporary variable `t1`. Note that we also obtain some type information from VEX. The statement `Put(8) = t8` writes the value from `t8` into `AL`. The last line sets the `EIP` register to the next instruction. Figure 4 shows the corresponding mappings to S-Expressions, where we can clearly see emerging similarity. The constructed assignment formulas are now ready for sampling basic blocks.

It is worth mentioning that we observed some unreliable behavior for floating point operations when using VEX statically. In fact, we identified only a few dozen floating point operations in the binaries that we analyzed, so we chose to replace floating point operations with NOPs. Again, it is only a matter of engineering to add support for floating point operations to our system.

B. Extracting Semantics: Sampling

To grasp the semantic I/O behavior of a basic block, we evaluate its formulas point-wise. More specifically, we generate random vectors with elements from the range $[-1000, 1000]$ and use them as input for the formulas. We found this space to be sufficiently large to avoid random output collisions, while being small enough to cover many possible inputs in the sample range. We used the same sequence of random input to evaluate all formulas to ensure that the computed outputs are comparable across formulas. We then create unique I/O pair representations by computing a 64-bit CRC checksum of the input length, the inputs, and the output value. Recall that our approach needs to work across architectures. As register names do not match across architectures, we exclude the output's name from the checksum computation.

We have to ensure that our sampling is robust to the order of variables. Let us consider the two formulas $a := b - c$ and $a := c - b$. They are the same, apart from the order of variables. However, with the inputs $(b = 1, c = 2)$ and $(b = 3, c = 5)$ their respective outputs are $(-1, -2)$ and $(1, 2)$, which does not reflect the similarity of the formulas. We thus also use the permutations as inputs, which are $(b = 2, c = 1)$ and $(b = 5, c = 3)$ in the example, and will obtain the

outputs $(-1,-2,1,2)$ and $(1,2,-1,-2)$, at which point we can observe similarity. Regardless of the CPU architecture, most basic blocks' formulas have only a few input variables. In `BusyBox`, less than 0.03% of the formulas have more than four input variables, so we can safely limit sampling to those formulas with at most four variables.

Another important detail relates to multiple layers of memory indirections (pointers to pointers). Since ARM and MIPS have a much larger number of registers than x86, there is naturally more register spilling necessary on x86. In our aggregated formulas, we observed many cases where the x86 version used a memory location as input or output, and ARM or MIPS simply used a register. However, a register use has only one input (the value it contains), while a memory indirection has two (the address specifying the location for indirection and the value at that location). Since we only compare formulas that have the same number of inputs, memory indirection thus effectively disrupted our formula comparison. Normally, to provide consistent memory (i. e., if an address is calculated twice in a formula, it references the same value), we would have to track the calculated addresses of nested memory indirections. Thus, for `EAX := [[EBX]]`, we sample a value for the register `EBX`, then for the memory location `[EBX]` and finally for the memory location `[[EBX]]`. However, for sampling, the final result of the computation is a value from memory (regardless of the layers of indirection), such that it suffices to track just one level of indirection. Effectively, we always provide an input variable for each indirection, if their address formula differed. That way, both an indirection and a register use account for only a single input.

C. Semantic Hash

We now have a list of I/O pairs, given as CRC checksums, for each basic block. Determining the similarity based on these I/O pairs would be expensive, both in respect to storage (due to the large number of necessary samples) and computation time. Therefore, we use the MinHash algorithm, which combines the I/O pairs of a basic block into a locally-sensitive hash. MinHash works as follows: it applies a (non-cryptographic) hash function to each element in the list and stores the minimal hash value among all I/O pairs. To compute the MinHash, this process is repeated with a variety of different hash functions. The more hash functions are used, the better the MinHash converges against the Jaccard index [5]. The purpose of the hash function is to randomly select an estimator for set-similarity. The similarity between two MinHashes averages over $i = 0...n$ such estimators:

$$sim(mh_1, mh_2) := |\{mh_1[i] = mh_2[i]\}|/n. \tag{1}$$

The expected error for n hash functions can be estimated to $\mathcal{O}(1/\sqrt{n})$ with a Chernoff bound [44]. We use 800 hash functions, which leads to an error of about 3.5%.

For our evaluation, we used an affine hash function of the form $h(x) := ax + b \bmod p$ with random 64-bit coefficients, where a prime modulus p guarantees that all coefficients are generators. To improve performance, we simulate further hash functions by transforming the output of the real hash function with rotation and XORing:

$$t(h(x)) := rotate(h(x), a) \oplus b. \tag{2}$$

The transformation changes the order of elements and therefore the selected minimum, which suffices for MinHashing.

We implemented two improvements of the standard MinHash algorithm. First, we compute multiple MinHashes per basic block, which we denote as *Multi-MinHash*. We do so by splitting the formulas into groups according to the number of input variables per formula and computing one MinHash per group. Later, we solely compare MinHashes for the same number of variables and compute the overall similarity by weighting the individual MinHashes by the number of formulas with the specific number of variables in the respective basic blocks. Thus, to compare two basic blocks, we compute

$$\frac{\sum_i s_i \cdot (w_i + w_i')}{\sum_i (w_i + w_i')}, \tag{3}$$

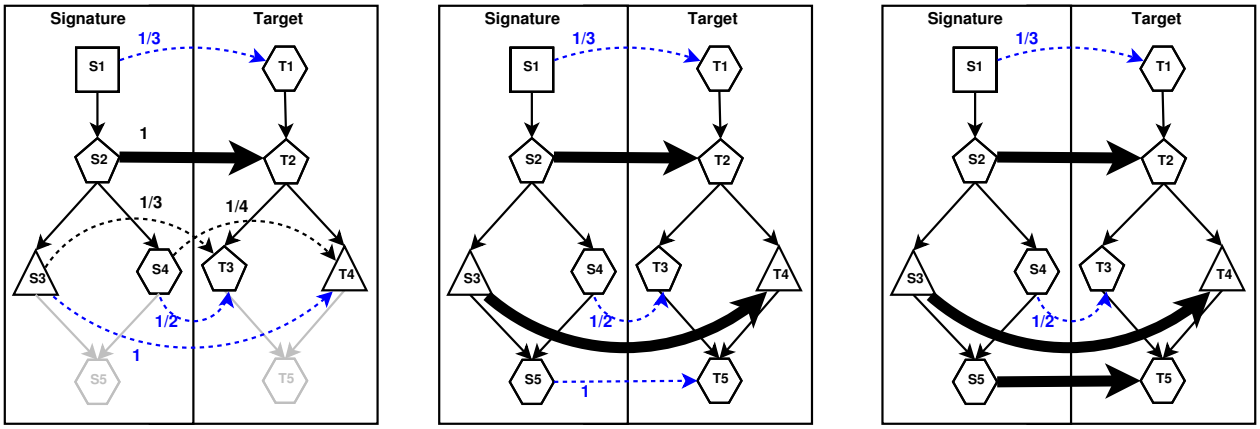
where s_i is the similarity of the formulas with i variables, w_i and w_i' the number of formulas with that number of variables in the firstand respectively the second, basic block. Multi-MinHash thus solves the issue that formulas with only a few samples (e.g., very few or no inputs) would be underrepresented in the hash value.

Second, we do not only store the smallest hash value per hash function, but the k smallest hash values—which we denote as *k-MinHash*. This modification allows us to estimate the frequency of elements in the multi-set of I/O pairs to some extent, i. e., we can recognize if a basic block has multiple equivalent formulas. Since deterministic hash functions map equal values to equal outputs, one cannot substitute k-MinHash against a MinHash with a larger number of hash functions. However, k-MinHash can be trivially combined with Multi-MinHash to benefit from both concepts. When referring to k-MinHash in the evaluation, we implicitly mean Multi-k-MinHash for $k = 3$.

D. Bug Signature Matching

Given a bug signature and a target program, we have to find code in the target program that is similar to the bug signature. To this end, we first iterate over the basic blocks of the bug signature and compare them individually against every basic block in the target program according to their MinHash similarity. For each basic block in the bug signature, we sort the resulting similarities, which results in a list of promising initial candidates for a full bug signature match. Then, we try to broaden the best b candidates with our *Best-Hit-Broadening* (BHB) algorithm, which computes the similarity of two graphs of basic blocks.

BHB works as follows: Given a pair of starting points (a basic block from the signature and its corresponding matching candidate in the target program), it first explores the immediate neighborhood of these basic blocks along their respective CFGs (Figure 5a). When doing so, it strictly separates forward and backward directions. After finding a locally-optimal



(a) First BHB round with the initial starting point and its candidate match (annotated with the bold arrow).

(b) After the first BHB round, another pair of BBs is matched and the lower two nodes are now adjacent.

(c) After the third step, three pairs are matched. There are no further neighbors and the other two matches are trivial.

Fig. 5: BHB example in three steps: bug signature on the left, target program on the right side. The difference between two basic blocks is visualized by the different numbers of edges (n) of the shapes. The similarity is then calculated as $\frac{1}{1+n}$.

matching among the newly discovered neighborhood nodes with a matching algorithm, it picks the basic block pair with the maximum similarity. That is, BHB broadens the already-matched basic block pairs (Figure 5b). The broadening is greedy, avoiding expensive backtracking steps. This process is repeated (Figure 5c) until all basic blocks from the bug signature have been matched. In the end, BHB computes the overall matching similarity as the average similarity of all matched pairs, which is ≈ 0.77 in Figure 5. BHB is then invoked for the other $b-1$ matching candidates. The end result of our bug signature search is a sorted list of BHB similarities, revealing those matched code parts in the target program that are most similar to the bug.

Listing 1: Best-Hit-Broadening Algorithm

```
1  BHB(sp_i ∈ SP, sp_t ∈ C_i, Sig)
2    M_S = [], M_T = []
3    PQ = (sim(sp_i, sp_t), (sp_i, sp_t))
4    while(PQ ≠ ∅)
5      P_S, P_T := PQ.pop()
6      M_S += P_S, M_T += P_T
7
8      D_p := sim_mat(P_S.pred \ M_S, P_T.pred \ M_T)
9      M̃_p := Hungarian(D_p)
10     PQ.add(sim(P_i, P_j), (P_i, P_j)), ∀(P_i, P_j) ∈ M̃_p
11
12     D_s := sim_mat(P_S.succ \ M_S, P_T.succ \ M_T)
13     M̃_s := Hungarian(D_s)
14     PQ.add(sim(P_i, P_j), (P_i, P_j)), ∀(P_i, P_j) ∈ M̃_s
15
16   dist := Σ_{i=0}^{|M_S|-1} sim(M_S[i], M_T[i])
17   return dist / |Sig|
```

A formal description of the algorithm is provided in Listing 1. As input, BHB accepts a basic block from the signature ($sp_i \in SP$) and a similar basic block from the target program ($sp_t \in C_i$). BHB keeps track of blocks that were already matched to each other (M_S from the bug signature and M_T from the target program) in a parallel list. Lastly, PQ is a priority queue of potential basic block matches, sorted by their similarity in decreasing order. That is, *pop* retrieves the basic block pair that is the current best match of the yet unmatched pairs. Note that PQ only includes pairs for which both basic blocks are *adjacent* to the basic blocks that have been matched so far. Initially (line 3), it only contains the starting point from the signature and its similar counterpart in the target program.

In lines 6 and 7, the algorithm takes the most similar pair and adds them to the set of matched basic blocks M_S and M_T. In lines 9 and 13, BHB computes the similarity matrix between blocks in the bug signature and all blocks in the target program that are adjacent to P_S and P_T, respectively. At this point, it finds a locally-optimal matching between those adjacent nodes to decide where the broadening should continue (lines 10 and 14). A so-called *matching algorithm* finds such a mapping between the left and right side of the graph, where the sum of all mapped similarities is maximal. We chose the *Hungarian method* [12], an algorithm with runtime $\mathcal{O}(n^3)$ in the number of nodes. Because we are only matching neighbors of the current signature basic block with the neighbors of the matching candidate in the target program, and do not backtrack, n is quite small. The overall runtime is thus dominated by the basic block-wise similarity metric.

The block pairs of that ideal mapping are added to the queue PQ, including their similarity, which is computed with the MinHash similarity (see Section III-C). BHB completes if the queue is empty, i. e., all basic blocks in the bug signature have been matched to a basic block in the target program (or no matching is possible anymore). Finally, in line 17, BHB computes the similarity between the bug signature and the target program by summing up the similarities for each matched block pair, averaging over the size of the bug signature.

To determine how many candidates b to examine, we

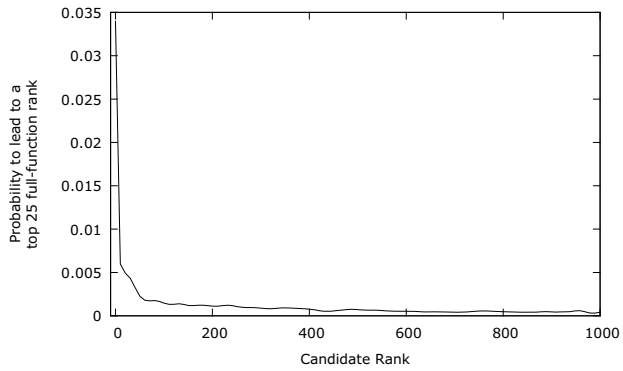


Fig. 6: Probability of the x-th candidate per basic block in the signature leading to one of the top 25 function matches.

analyze the probability of the x-th candidate per basic block in the signature leading to one of the top 25 function matches. Fig. 6 shows that the best function matches usually stem from the best individual candidate matches. For this paper, we choose to investigate only the first $b = 200$ candidates with a full-blown invocation of the BHB algorithm, which offers a reasonable trade-off between accuracy and performance.

IV. Evaluation

This section gives an empirical evaluation of our approach: First, we systematically evaluate our code comparison metric in a binary, cross-architecture setting. Second, we apply it to our use case: finding bugs using a bug signature.

The experiments were conducted on an Intel Core i7-2640M @ 2.8GHz with 8GB DDR3-RAM. To ease comparability, they were performed with a single-threaded process. Our experiments include 60 binaries from three architectures and three compiler versions (x86, ARM, MIPS; all 32 bit) (`gcc` v4.6.2/v4.8.1 and `clang` v3.0). While our focus was on Linux, we encountered different core libraries, especially for router software (`DD-WRT`/`NetGear`/`SerComm`/`MikroTik`). Note that we also compared binaries across Windows and Mac OS X, i.e., we covered all three major OSes.

Our approach allows us to find (sub-)function code snippets at any granularity. Still, the following experiments mostly show function-level comparisons, as function-level signatures can be chosen automatically. If we manually excluded some blocks, we would need to justify that choice. Instead, we decided to use a fair and reproducible test case: function-level signatures. However, in practice, analysts can choose entire functions or only parts of a function as a signature.

A. False/True Positives Across Architectures

In this experiment, we aim to match all functions of one binary A to all functions of another binary B, which measures how generic our solution is in matching similar code. The two binaries stem from the same source code base, and thus principally contain the same code, but were compiled for different architectures. For each function in A, we compute

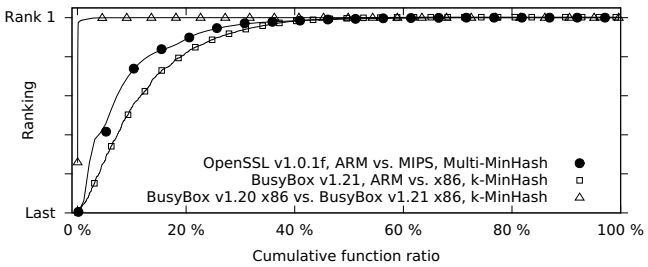


Fig. 7: Ranks of true positives in function matching for `OpenSSL` and `BusyBox`. Other architecture-combinations are similar and were omitted.

the similarities to all functions in B, resulting in a list of functions from B sorted by their similarity to the searched-for function. We then plot the rank of the correct counterpart of the function in A, i.e., the position in the sorted list, where the most similar match is at rank 1. We derive the ground truth of the function mappings based on the (identical) function names in both binaries. Note that we use the symbolic information for this mapping only and not in the similarity metric.

First, we evaluate our approach on a single architecture. We compared two x86 binaries of very similar, but not identical versions of `BusyBox` (v1.20.0 and v1.21.1). Figure 7 shows a cumulative distribution function (CDF) on the ranks of all functions of this experiment. A perfect method would result in a straight line, i.e., all correct functions are at rank 1. In this single-architecture comparison, we could perfectly match 90.4% of the functions (rank 1) and had close matches for 97% (ranked within the top 10). Some of the mismatches may also be caused by slight changes between the two `BusyBox` versions. In any case, this experiment estimates a baseline for the best quality that we can expect in the cross-architecture setting.

In the second experiment, we took two binaries of the same program (`BusyBox` v1.21.1 and `OpenSSL` v1.0.1f) compiled for different architectures. Figure 7 shows that we can rank about 32.4% of the functions in `BusyBox` (ARM to x86) at rank 1, 46.6% in the top 10 and 62.8% in the top 100. For `OpenSSL` (ARM to MIPS) we even reach 32.1% for rank 1, 56.1% in the top 10 and 80.0% in the top 100. Bad rankings mainly resulted from partial structural differences in the binaries. For example, the MIPS version of `BusyBox` has 65 more functions, which suggests different inlining, and almost 13% more basic blocks, which alters the CFG of many functions. Our metric is sensitive to the CFG and the segmentation of the basic block, which we found to be potentially problematic especially for smaller functions. However, the chance that these less-complex functions contain vulnerabilities is also significantly lower [26].

Still, for the majority of cases, Figure 7 shows that our semantic basic block hashes actually provide a reasonable degree of similarity across architectures. In many cases, the CFGs are sufficiently similar across architectures to allow for meaningful comparisons, which means that both the CFG structures and

the separation of basic blocks are largely preserved.

We do not consider function matching as the primary use case of our approach. Our metric punishes non-matched basic blocks in the signature, but not in the target function. Thus, a good match for a small part of the function is ranked higher than a mediocre, but exhaustive match. However, even with this metric, the results could be improved, e. g., by fixing good function matches and thereby successively eliminating candidates for other function matches.

B. False/True Positives Across Compilers/Code Optimization

The previous example has shown that function matching is possible even if binaries are compiled for different architectures. Next, we systematically evaluate how the choice of compiler and optimization level influence the accuracy of our algorithm. We chose the largest `coreutils` programs (measured at O2, which is the default optimization level, e. g., in Debian and for automake). As opposed to all other experiments in this paper, we *compiled these programs ourselves* on x86 with three different compilers (`gcc` v4.62, `gcc` v4.81 and `clang` v3.0) and in four optimization levels (O0-O3).

True Positives: In a first experiment, we aim to systematically evaluate the true positive rate in the cross-compiler scenario. To this end, for each program, we compared all 12 binaries with each other (i.e., all 144 binary pairs) using function-wise matching. For example, we compare `cp` compiled with `clang` in O1 with `cp` compiled with `gcc` v4.62 in O2. Similarly to the previous experiment, we show the ratio of correct function matches. However, in contrast to the previous experiments, the resulting $9 * 144 = 1296$ pairs have to be visualized very densely, making individual CDFs impractical. Thus, in this experiment (and in this experiment only), we visualize a match as a true positive, if it is among the top 10 matches.

Figure 8 illustrates the results of this experiment in a matrix. Each dot in the nine cells illustrates the algorithm's accuracy, given one concrete pair of binaries of the same program. The twelve sub-columns and sub-rows per cell are divided as follows: columns/rows 1-4 are `clang` in O0-O3, columns/rows 5-8 are `gcc` v4.62 at O0-O3, and columns/rows 9-12 represent `gcc` v4.81 at O0-O3. The darker a dot, the more function pairs matched correctly (100% is black). Discussing each dot in Figure 8 is not possible due to space constraints, but we can generalize the following observations from this experiment: (1) The search results are symmetric, i.e., they do not significantly change if we, e.g., search from `gcc` binaries to `clang` binaries or vise versa. This is good, as the direction in which a search must be made is generally unknown. (2) Comparing programs compiled for O0 (i.e., no optimization) to binaries with any other optimization level significantly weakens accuracy. Luckily, programs are rarely compiled without any optimization in practice. (3) Binaries compiled by the different `gcc` versions have a higher similarity to each other than binaries created with different compilers. While cross-compiler results (i.e. `clang` vs. `gcc`) are worse than intra-compiler results, they still provide meaningful rankings. (4) Comparing binaries across different optimization levels (O1-O3) is typically possible with high accuracy. That is, more advanced optimization strategies introduced in later optimization levels (O2 and O3) do not severely harm the overall performance of our system.

False Positives: In a second experiment, we aim to measure false positives produced by our algorithm. In principle, the algorithm returns a list of functions with their similarities to the bug signature. As such, it is not straightforward to judge if the highest-ranked function is indeed a match, or is just the function that was least different from the bug signature. Thus, in general, evaluating false positives in our system is inherently difficult (if not impossible). Having said this, we acknowledge that there should be such an evaluation to illustrate the accuracy of our system. Thus, and only for the purpose of this experiment, we define a metric which judges if the highest-ranked function is a match or not. We chose to consider the highest-ranked potential function match as an actual match if its similarity is at least 0.1 higher than the second-ranked match (an admittedly arbitrarily-chosen threshold on a scale from 0–1). The intuition behind this heuristic is as follows: If no function matches, all functions will be more or less equally different, i.e., their similarity scores are more densely connected. However, an analyst might be misled by a highly ranked function with a low similarity score, if that function stands out from the others—that is, if its similarity is high in contrast to the next-best match. Again, note that we use this threshold for this false positive experiment only. It is *not* an integral part of our system and is also not used in the other experiments (see also the discussion in Section V).

To this end, we focused on the nine `coreutils` programs in a single, common compiler setting. We chose to use `gcc` due to its popularity (in v4.62) and the optimization level O2 (again, since it is often the default optimization level). We then compared each program with every other program, resulting in 81 program comparisons. Again, we tried to match all functions in program A with all functions in program B.

Figure 9 illustrates the similarity of two different programs. For each program, we compute the ratio of functions that have been supposedly erroneously matched with the other program. We excluded functions from the matrix that had the same names in two `coreutils` programs (i.e., code sharing between programs), as they would bias the evaluation. The cells represent the ratio of function matches, i.e., highest-ranked functions whose similarity score is significantly (the 0.1 threshold) higher than the second-ranked function. That is, on the diagonal line, in which we actually do expect matches, dark cells represent high true positive rates. On the contrary, in all other cells, where we do *not* expect matches, dark cells represent high false positive rates. The figure shows that even if we use such a crude mechanism to judge un actual matches, the false positive rate of our system is fairly low. We suffered seemingly many false positives when comparing functions of `ls` and `dir`. In fact, these programs are almost identical, leading to many (correct) matches. Since those programs had the same optimization level, whereas the main diagonal

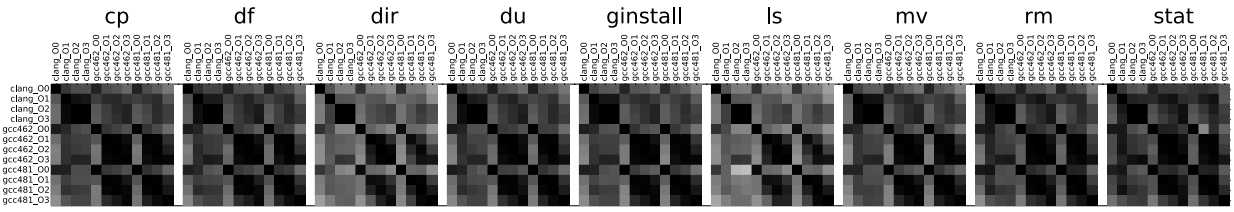


Fig. 8: True positive matrix for the largest `coreutils` programs, compiled with three different compilers and four different optimization levels. Darker colors represent a higher percentage of correctly matched functions (see Figure 9 for color scale).

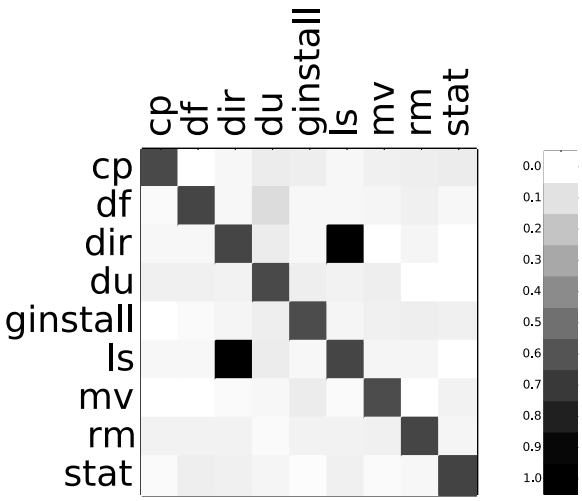


Fig. 9: Fraction of strong matches for the largest `coreutils` programs. Darker colors represent a higher percentage of top-1 ranks, which exceed the threshold of 0.1 similarity difference to the top-2 rank. On the main diagonal, these matches are true positives; in all other cells they reflect false positives.

averages over all optimization levels, the matching results are actually even better for this special case.

C. Bug Search in Closed-Source Software

In this section, we evaluate several case studies of recent prominent cross-architectural vulnerabilities. For example, we find the *Heartbleed* bug on desktops as well as on mobile devices and router firmware images. Similarly, we find `BusyBox` bugs, which are part of closed-source bundles of embedded devices (e.g., home routers) across multiple architectures. Finally, we identify a bug and a backdoor in closed-source firmware images.

Our method is targeted towards fine-grained, sub-function code similarity. A bug signature should hold the most discriminatory parts from the buggy binary code, which may include context that does not strictly belong to the bug. We could easily generate such signatures automatically from source code information (see Section II-B). While this allows one to tweak bug signatures to achieve optimal results, we chose not to do so in our experiments to ensure reproducibility and comparability

TABLE I: Ranks of functions vulnerable to *Heartbleed* in `OpenSSL` compiled for ARM, MIPS and x86, in `ReadyNAS` v6.1.6 (ARM) and `DD-WRT` r21676 (MIPS) firmware. Each cell gives the ranking of the TLS/DTLS function.

	Multi-MH		Multi-k-MH	
From → To	TLS	DTLS	TLS	DTLS
ARM → MIPS	1;2	1;2	**1;2**	1;2
ARM → x86	1;2	1;2	1;2	1;2
ARM → `DD-WRT`	1;2	1;2	1;2	1;2
ARM → `ReadyNAS`	1;2	1;2	1;2	1;2
MIPS → ARM	2;3	3;4	1;2	1;2
MIPS → x86	1;4	1;3	1;2	1;3
MIPS → `DD-WRT`	1;2	1;2	1;2	1;2
MIPS → `ReadyNAS`	2;4	6;16	1;2	1;4
x86 → ARM	1;2	1;2	1;2	1;2
x86 → MIPS	1;7	11;21	1;2	1;6
x86 → `DD-WRT`	70;78	1;2	5;33	1;2
x86 → `ReadyNAS`	1;2	1;2	1;2	1;2

of our work. Instead, we declare the entire function containing a bug as our bug signature.

1) OpenSSL/Heartbleed: In April 2014, the critical *Heartbleed* bug (CVE-2014-0160) was fixed in the `OpenSSL` cryptography library. Since `OpenSSL` is an integral part of many TLS implementations, this security-critical bug is widely deployed, including many closed-source software applications.

The *Heartbleed* bug allows an attacker to perform an out-of-bounds read, which is, due to the handled key material and `OpenSSL`'s built-in memory management, highly security-critical. The bug can be triggered remotely by manipulating heartbeat (keep-alive) messages [34]. The vulnerable functions are `tls1_process_heartbeat` (TLS) and `dtls1_process_heartbeat` (DTLS) in v1.0.1a-f.

We extracted one bug signature for each of these two functions from v1.0.1f in an automatic fashion, which required only the vulnerable function's name and its starting address. Again, we stress that more specific signatures might fit better into our tool's niche of fine-grained code comparison. However, we chose not to manually refine the signature in order to avoid deceptive and possibly over-fitted signatures.

We use these signatures to find the vulnerable functions in `OpenSSL` binaries that we compiled for x86, MIPS, and ARM. In addition, we also search in two real-world occurrences of vulnerable `OpenSSL` libraries in firmware images: the Linux-based router firmware `DD-WRT` (r21676) compiled for MIPS and a NAS device (Netgear `ReadyNAS` v6.1.6) with an ARM processor [9], [31]. We took the firmware images provided on the project websites, unpacked them and searched

TABLE II: `make_device` ranks in `BusyBox` for Multi-MH.

From → To	*v1.19.0 ARM*	*v1.19.0 MIPS*	*v1.19.0 x86*	*v1.20.0 MIPS*	*1.20.0 x86*
v1.20.0 ARM	1	1	1	1	1
v1.20.0 MIPS	6	1	4	1	1
v1.20.0 x86	1	2	1	1	1

TABLE III: `socket_read` ranks in firmware for Multi-MH.

From → To	DGN1000	DGN3500	DM111Pv2	JNR3210
DGN1000	-	1	2	1
DGN3500	1	-	1	1
DM111Pv2	2	2	-	1
JNR3210	1	1	1	-

with the automatically generated signatures.

Table I shows the ranks of the vulnerable functions in `OpenSSL`, where "From" indicates the architecture that we used to derive the bug signature, and "To" indicates the architecture of the target program. Each cell contains the ranking of both vulnerable functions (heartbeat for TLS and DTLS). Table I shows that the combination of semantic hashes and BHB works well, giving almost perfect rankings. For example, the highlighted cell shows that our algorithm ranks the vulnerable TLS and DTLS functions in the MIPS binary at rank 1 and 2, respectively, when searching with the corresponding signature from an ARM binary.

Again, we observe that the improved semantic hash (k-MinHash) improves accuracy. E. g., for `OpenSSL` x86 vs. MIPS with the DTLS-signature, we noticed improved ranks, because seven basic blocks had duplicated formulas, which Multi-MinHash cannot detect (see Section III-C).

Searching from and to ARM and x86 works almost perfectly, while combinations with MIPS sometimes work a little less well (e. g., for `OpenSSL` x86 vs. `DD-WRT`). Usually, due to its RISC instruction set, MIPS has either more basic blocks (about 13% more for `BusyBox` ARM vs. MIPS) or more formulas (1.42 times for `OpenSSL` MIPS vs. x86).

2) BusyBox Vulnerabilities: In late 2013, a bug was discovered in `BusyBox` versions prior to v1.21.1, where subdirectories of `/dev/` are created with full permissions (0777), which allows local attackers to perform denial-of-service attacks and to achieve local privilege escalation (CVE 2013-1813 [38]). The affected function `build_alias()` is inlined into the `make_device()` function, which complicates matters for techniques relying on function equivalence. Table II shows that our technique succeeds in identifying similar code in the context of inlining, where, in contrast to function matching, only a subarea of a function should be matched. Note that the ranking is not perfect when searching in ARM code, where `make_device()` has only 157 basic blocks, with a bug signature from MIPS code (183 basic blocks).

3) RouterOS Vulnerability: In 2013, a vulnerable function in MikroTik's `RouterOS` v5 and v6 was found in the SSH daemon. Due to a missing length check in the method `getStringAsBuffer()`, the attacker can trigger a segmentation fault. This allows a remote heap corruption without prior authentication, which can be leveraged to arbitrary code execution [23]. RouterOS is available for both MIPS and x86. Using the vulnerable function for either architecture as a bug signature, we reliably found the vulnerable function for the other architecture at rank 1. We obtained similar results for all hashing methods, but omit the table due to space constraints.

4) SerComm Backdoor: Lastly, we show that our approach can also be used in other contexts, such as finding known backdoors. To demonstrate this, we search for a known backdoor in SerComm-powered firmwares [41], which opens a shell once it receives a special packet on TCP port 32764. Hardware by SerComm is used by multiple router manufacturers (like Cisco, Linksys and Netgear). We defined the `socket_read()` function as a bug signature and searched for similar backdoors in MIPS-based Netgear firmware (`DGN1000`, `DGN3500`, `DM111Pv2`, `JNR3210`). Table III shows that we find the backdoors reliably in all combinations.

5) libpurple Vulnerability: Up to this point, we only showed case studies with full functions as signatures, even though we do not consider this our primary use case. Our main motivation for doing so lies in the fact that any hand-crafted signature has to be justified, so as not to be dismissed as tweaked for the particular example. Nevertheless, we feel the need to highlight that, especially for bug search, full function matching (as, e. g., done with BLEX [11]) is not sufficient.

CVE-2013-6484 documents an attack that allows an attacker to crash versions prior to v2.10.8 of `Pidgin`, a popular instant messenger. Both the `Windows` version of `Pidgin` and its `Mac OS X` counterpart `Adium` (v1.5.9) suffer from this vulnerability. In this example, the vulnerable function in `Pidgin` contained many basic blocks of other inlined functions, whereas `Adium` did not inline them. Consequently, the `Pidgin` function had 25% more basic blocks.

Intuitively, both functions differed significantly, and particularly using the larger function as a bug signature may introduce problems. Indeed, our tool—presumably similarly to purely function-wise matching approaches, such as BLEX—could not achieve good ranks for full function signatures. From `Windows` to `Mac OS X` and *vice versa* we achieved rank #165 and rank #33, respectively.

However, when choosing ten basic blocks by hand, we achieved rank #1 in both cases. We did so in a methodical way: We included all basic blocks from the function start through the vulnerability, while avoiding the early-return error states.

D. Unpatched vs. Patched Code

We noticed that a patch typically introduces few changes to a vulnerable function. This causes our approach to also identify patched code parts as similar to the bug signature, effectively causing false positives. In an experiment to tackle this issue, we created two bug signatures: in addition to the one for the vulnerable function, we also defined a signature for the patched function. Although both signatures will match reasonably well, intuitively, the latter signature has a higher similarity for patched code than the original bug signature. Tab. IV shows that for the bugs in `OpenSSL` and `BusyBox`,

TABLE IV: Unpatched/patched signature similarities with Multi-MinHash in `OpenSSL` and `BusyBox`.

From → To	Rank	Similarity
`OpenSSL` x86 → `ReadyNAS` (ARM)		
v1.0.1f (unpatched) → v6.1.6 (unpatched)	1	0.3152
v1.0.1g (patched) → v6.1.6 (unpatched)	1	0.1759
v1.0.1f (unpatched) → v6.1.7 (patched)	1	0.3021
v1.0.1g (patched) → v6.1.7 (patched)	1	0.3034
`BusyBox` ARM → `BusyBox` x86		
v1.20.0 (unpatched) → v1.20.0 (unpatched)	1	0.2676
v1.21.1 (patched) → v1.20.0 (unpatched)	1	0.1689
v1.20.0 (unpatched) → v1.21.1 (patched)	1	0.1658
v1.21.1 (patched) → v1.21.1 (patched)	1	0.2941

TABLE V: Runtime in minutes in the offline-phase.

Entity	#BBs	IR-Gen.	Multi-MH	k-MH
`BusyBox`, ARM	60,630	28	80.3	1522
`BusyBox`, MIPS	67,236	35	86.9	1911
`BusyBox`, x86	65,835	51	85.0	1855
`OpenSSL`, ARM	14,803	8	16.3	349.7
`OpenSSL`, MIPS	14,488	9	16.3	434.0
`OpenSSL`, x86	14,838	12	20.2	485.2
Normalized Avg.	10,000	6.2	12.5	279.9

either bug signature (patched or unpatched) ranked both software versions at rank 1. However, in all cases, the unpatched version is more similar to the "buggy" bug signature, and vice versa (considerably in three cases, marginally in one). This could give valuable information to an analyst deciding whether he is looking at a false positive.

E. Performance

Lastly, we evaluate the performance and the scalability of our system. Our approach has three computational phases (IR generation, semantic hashing and signature search) for which we will give separate performance evaluations.

IR generation: The IR formulas only need to be computed once per binary. The runtime to describe the I/O behavior increases linearly with the number of basic blocks, but varies with each basic block's complexity. If it consists only of a jump, this step is considerably faster than for a basic block with many instructions. Since both types of basic blocks regularly appear, Table V shows the average timings achieved in real-life programs, showing that formulas can be created in less than an hour for all programs under analysis.

Semantic hashing: In the case of hashing, the number of formulas in a basic block and their respective number of input variables dominates the runtime. We thus limited the number of samples to 3000, whereas we used fewer samples for formulas with fewer input variables. The sample set is sufficiently large to capture semantic differences and the accuracy did not significantly improve for larger sets. At the same time, as shown in Table V, sampling and MinHash of real-life programs scales. Clearly, the higher accuracy of k-MinHash (see Table I) comes at the price of performance degradation of a factor of ≈ 22.5 on average. Again, note that this computation is a one-time-only effort and can be parallelized trivially down to a basic block level.

Signature Search: The two previous steps represent one-time costs and do not influence the overall performance as much as this third step: the actual search phase. Recall that we initially search for promising candidates for each basic block in a signature by comparing it against all basic blocks in the target program. The runtime thus increases in a linear fashion for both the number of basic blocks in the signature (usually a few dozen) and the number of basic blocks in the target program (usually in the order of 10^4).

TABLE VI: Runtimes of the signature search.

	# BBs		Runtime	
	Sig	Target	Multi-MH	k-MH
`BusyBox` v1.20.0				
ARM → MIPS	157	70,041	230.6s	754.0s
ARM → x86	157	66,134	197.7s	644.4s
`OpenSSL` v1.0.1f				
MIPS → ARM	25	14,803	14.2s	46.8s
MIPS → x86	25	14,838	14.4s	46.9s
Normalized Avg.	1	10,000	0.3s	1.0s

Table VI gives exemple runtimes for the signature search in various scenarios. Typically, the runtime is in the order of minutes for the real-world use cases. In the worst case, it took about 12.5 minutes to search for bugs with k-MinHash in a MIPS-based `BusyBox`. The evaluations show that the complexity of the signature has a high impact on the runtime.

V. Discussion

This section discusses some of the remaining challenges of our system, most of which we plan to address in future work.

A. Vulnerability Verification

One challenge that we already touched upon in Section IV is the fact that our approach cannot verify that the code part that was found is actually vulnerable. Such an automatic verification would be ideal, but surely is a research topic in itself and is outside the scope of this work. The ranking produced by our system gives an ordered list of functions that have to be manually inspected by an analyst to verify if those functions are actually vulnerable. This way, the manual process is greatly reduced, as the analyst will oftentimes find the vulnerability in the top-ranked functions.

In case a binary does *not* contain the bug that was searched, the ranking scheme still gives a list of functions. Naturally, an analyst would then fail to find vulnerable functions, even when inspecting all ranked functions. Ideally, we could give some indication if there is a reasonable match at all. This could, e.g., be based on the similarity score, which represents the semantic similarity between the bug signature and a function: If the functions ranked first have "low" similarities, this suggests that even the best hits are not vulnerable.

In future work, we will investigate whether we can expand our scheme with "similarity thresholds" that can separate potential matches from non-matches in the ranking. The heuristics that we used in Section IV-B are only a first step. A better, more reliable mechanism to determine actual matches will allow for further use cases of our work, such as large-scale binary searches to identify license violations.

B. False Negatives

A few challenges could cause false negatives in our approach, i.e., it would miss actual vulnerabilities. Note that we explicitly exclude obfuscated binaries from our scope.

We have shown that although common off-the-shelf compiler optimizations may weaken detection results, the extent of this is limited. For example, our method relies on basic blocks being split in a similar way and does not tolerate substantial changes to the CFG in the important (i.e., buggy) parts of the code. We showed that these assumptions are indeed met in many practical examples, even across processor architectures.

One of the core problems of comparing software at the binary level is that its representation (even on a single architecture) heavily depends on the exact build environment that was used to compile the binary. Varying compilers have different strategies to translate the same source code to binaries, as they implement different optimization techniques (and levels). In the following, we briefly discuss some of the most common optimization techniques and compiler idiosyncrasies and how our system tackles them.

1) Register spilling: The number of general-purpose registers varies between the CPU architectures: x86 features 8 such registers, legacy ARM 15, and MIPS 32. Architectures with fewer registers typically need to spill registers more frequently, i.e., they have to move data between memory and registers. Naïvely, this has an effect on the I/O pairs of basic blocks, and thus complicates comparison between different architectures. As described in Section III, we successfully addressed this issue by flattening nested memory indirections.

2) Function Inlining: Function inlining heavily changes the CFG of a program and may thus become problematic for our approach. However, if the bug signature covers the inlined function, then our approach can still find the code parts. In fact, the larger context in which the vulnerable code parts are embedded is not relevant for our approach, as long as the sub-CFGs remain similar. Thus, as also demonstrated with `BusyBox`, we can find buggy code that has been inlined (cf. Section IV-C2). Things become slightly worse if the bug signature was derived from a function that included inlined functions and if the target programs do not show such inlining. But even then, using the multiple initial basic block matches in our BHB algorithm, our approach can likely find the multiple counterparts in the non-inlined target program.

3) Instruction Reordering: Compilers may reorder independent computations to enhance data locality. Reordered instructions in a basic block change the syntactic assignment formulas in the IR. However, ultimately the I/O pairs are the same, as otherwise the semantics of the basic block would have changed. By comparing the code semantics, our system thus tackles reordering for free.

4) Common Subexpression Elimination: When compilers eliminate common subexpressions, such as when optimizing the expression $(x * y) - 2 * (x * y)$, neither the output value nor the number of inputs change. However, our system will create additional formulas for additional temporary variables that the compiler may use.

5) Constant Folding: Our approach can easily deal with constant folding, such as $2 + 4$ becoming 6 at compile time. Either way, using sampling, the output variables will have equal values. This is an additional advantage of comparing semantics instead of syntax.

6) Calling Conventions: Our approach is largely independent of calling conventions. As we abstract from concrete register names, to us it is not important which registers (or stack/memory addresses) are used to pass registers or to return results. It would be problematic when comparing the syntax of the IR representations, but when hashing the sampled I/O pairs, our approach completely ignores register names.

This list of compiler idiosyncrasies is incomplete, but covering all of them would go beyond the scope of this discussion. We have to acknowledge that some optimizations modify the CFG (e.g., loop unrolling, dead code elimination) and may become problematic if they affect the code parts of the bug signature. However, our evaluation has shown that our system performed well in a realistic setting and implicitly covered many of the optimization cases. Recall that most of the experiments conducted in Section IV were based on real-world binaries that were compiled by various vendors. That is, we did *not* self-compile these binaries in a homogeneous setting. Instead, our algorithm worked well for heterogeneous build environments (different compilers, optimization strategies, etc.), underlining the usability of our approach in a practical setting.

C. Scalability

We performed all experiments in a single process/thread and did not use any parallel computing to speed up the experiments. The algorithm with the best results, k-MinHash, degrades performance quite significantly. This may become problematic when searching multiple signatures in a large database of binaries. A solution could be to run the computationally cheaper algorithms (e.g., Single-MinHash) first and then re-process the high ranks with k-MinHash. Moreover, most computations can be scaled up with straightforward parallelization. The unification/sampling/hashing phases can run in parallel on a basic block level, which would reduce the runtime by orders of magnitude with commodity servers.

In addition, note that the most compute-intensive parts of our approach are one-time operations. That is, translating the binaries, sampling and hashing has to be performed only once per binary under analysis. Only the matching phase needs to be run once per search for a bug signature—a process that can also easily run in parallel.

VI. Related Work

To the best of our knowledge, we are the first to propose a strategy for comparing binary code across different architectures. Our work thus differs from other works that have a similar use case (bug finding) and it is therefore hard to directly compare our cross-architecture bug search with other works that operated under less challenging conditions (e.g., with available source code or only for a single architecture). This is also the reason why we did not compare our evaluation to existing approaches, simply because there is no other approach that operates on multiple architectures. In the following, we review works that also aim to find code similarity or bugs, albeit using quite different preconditions.

A. Code Similarity

A first line of research has proposed methods to find code clones on a source code level. CCFINDER [22] can find equal suffix chains of source code tokens. DECKARD [18] leverages abstract syntax trees and local sensitive hashes to find similar subtrees of source code. Yamaguchi et al. [42] broaden this concept with methods from text mining to not only re-find the same bug, but in the best case extrapolate vulnerabilities by finding close neighbors, or even to find missing checks through anomaly detection. The latter was also studied independently by Gauthier et al. [14]. Lastly, REDEBUG [17] is a highly scalable system to find unpatched code clones in many popular programming languages. As opposed to our system, these approaches require source code and thus cannot aid code search in binary software, especially not for closed-source software.

Lacking symbolic information complicates the process of comparing code at the binary level. Early approaches to find binary code similarity rely on sequences of occurring calls [1] or assembly K-grams [30]. Other approaches follow strategies to compare the semantics of code. BINHUNT [13] uses symbolic execution and a theorem prover to show that basic blocks are semantically equivalent, but suffers from performance bottlenecks. A follow-up project, IBINHUNT [29], augments BinHunt with taint analysis to reduce the number of possible matches. Not only are both works tailored towards equivalence instead of gradual similarity, which is problematic for bug extrapolation, they are both specific to x86, which makes them incomparable to our work. BINJUICE [24] translates basic blocks into syntactic equations (similar to our formulas) and uses the concatenated and hashed equations as one input to measure similarity between two basic blocks. However, this approach is assumed to fail across CPU architectures, as we have shown that formulas are syntactically different for each architecture. BINHASH [20] relies on the input/output behavior of equations, but also does not include steps to support multiple architectures. Furthermore, they cannot use the CFG, but instead have to aggregate the hashes on a function level. This precludes sub-function granularity, which is problematic for our use-case. EXPOSÉ [32], a search engine for binary code, uses a theorem prover to match binary functions, but operates on a function level and simply assumes a single calling convention (`cdecl`), which is impractical even on a single CPU architecture.

Concurrent to our work, Egele et al. proposed *Blanket Execution* (BLEX) [11], a system to match functions in binaries. They execute each function of a binary for a low number of execution contexts (according to the authors, three is sufficient) and save the function-level output. They also make sure to execute each of the functions' basic blocks at least once, simply by re-executing the function from the first basic block that was not yet covered. Note that the changes to the execution contexts propagate through the basic blocks and thereby only function-level output is considered. While BLEX can compare binaries compiled with different optimization levels, it can only perform function-level matching. Due to this fact, the number of entities which have to be compared is quite low in comparison to our work, which is why they did not have to use techniques like MinHashing. BLEX currently supports only x64 binaries, and extending its scope to other architectures would be tedious—both because BLEX is based on Intel's Pin and because it does not address many of the challenges of cross-platform binary comparisons (cf. Section II-C).

Our work has similar goals as TEDEM [33], which uses a notion of signatures to find bugs in binaries. We also perform a greedy search when broadening the signature, as our proposed Best-Hit-Broadening algorithm shows. However, the authors use tree-edit distance metrics, which severely slow down the matching process and do not capture all syntactical changes. Thus, TEDEM is not suitable for cross-platform comparisons.

Our approach overlaps with some ideas of these works, but has many novel aspects. We developed a cheap, semantic metric, which utilizes the CFG and is able to operate on sub-function granularity. Most importantly, we are the first to compare binaries across architectures by unifying binary code from multiple architectures to a common representation.

B. Identifying Previously-Unknown Bugs

We aim to identify bugs that are similar to well-known bugs. Orthogonal related work searches for previously-unknown bugs. To this end, some works rely on source code to find vulnerabilities. For example, in AEG [2], Avgerinos et al. suggest using preconditioned symbolic execution in order to explore exploitable program paths. Similarly, COVERITY [4] uses static analysis to reveal potential memory corruption vulnerabilities or data races; however, it fails to verify whether all preconditions for exploitation are actually met, and also does not cover all bug classes. In fact, the authors described why Coverity did not detect the Heartbleed bug [7].

Shankar et al. proposed using a type-inference engine to detect format string vulnerabilities [35]; a similar principle was followed by Johnson and Wagner [21]. Other approaches can identify new bugs even without access to source code. Arguably the first work in this area was Miller et al.'s proposal to use blackbox fuzzing to provoke program crashes [28]. Livshits and Lam use tainting to find vulnerabilities in Java bytecode [25]. Recently, Cha et al. introduced an efficient symbolic execution engine to find previously-unseen bugs at the binary level with MAYHEM [6]. In general, dynamic analysis approaches like fuzzing or Mayhem require an operable environment where the software is actually running. As illustrated by Zaddach et al. [43], this is far from easy on highly-specialized hardware such as embedded devices. In contrast, a purely static analysis like ours does not have to deal with the peculiarities of the actual hardware platform other than its CPU architecture. Moreover, many existing solutions are tailored to find only one class of bugs, such as control flow hijack vulnerabilities. As such, they are not generally suitable to find any class of bugs, as our system is. On the other hand, these tools can find previously-unseen bugs, while

our approach is focused on re-finding bugs in other binary software. We thus consider these works as complementary.

VII. Conclusions

In this paper, we showed that semantic binary code matching is possible across CPU architectures under reasonable assumptions that hold in practice. This advances prior research results that are restricted to comparing binary code of a single architecture. Our novel metric allows for fine-grained code comparison, which we successfully applied to identify real-world vulnerabilities in closed-source software.

The pervasiveness of the Heartbleed bug exemplifies the importance of identifying code parts in closed-source binary programs that contain a specific vulnerability. With the rise of closed-source software on other architectures (e.g., Windows RT and iOS on ARM, or numerous firmware binaries for MIPS), and re-use of potentially vulnerable code in such software, our approach can greatly assist in finding future vulnerabilities in binaries compiled for any architecture.

Acknowledgment

This work was supported by ERC Starting Grant No. 640110 (BASTION) and by the German Research Foundation (DFG) research training group UbiCrypt (GRK 1817).

References

[1] F. Ahmed, H. Hameed, M. Z. Shafiq, and M. Farooq. Using Spatio-temporal Information in API Calls with Machine Learning Algorithms for Malware Detection. In *ACM AISec*, 2009.

[2] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Symposium on Network and Distributed System Security (NDSS)*, 2011.

[3] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. Byteweight: Learning to recognize functions in binary code. In *USENIX Security Symposium*, 2014.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of ACM*, 2010.

[5] A. Z. Broder. On the Resemblance and Containment of Documents. In *IEEE SEQUENCES*, 1997.

[6] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing Mayhem on Binary Code. In *IEEE Symposium on Security and Privacy*, 2012.

[7] A. Choue. On Detecting Heartbleed with Static Analysis, Apr. 2014. http://tinyurl.com/hb-coverity.

[8] A. Costini, J. Zaddach, A. Francillon, and D. Balzarotti. A Large-Scale Analysis of the Security of Embedded Firmwares. In *USENIX Security Symposium*, 2014.

[9] DD-WRT. r21676, May 2013. http://tinyurl.com/ddwrt-21676.

[10] T. Dullien and R. Rolles. Graph-based comparison of executable objects. *SSTIC*, 5, 2005.

[11] M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security Symposium*, 2014.

[12] A. Frank. On Kuhn's Hungarian Method – A Tribute From Hungary. Technical report, Oct. 2004.

[13] D. Gao, M. Reiter, and D. Song. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *Information and Comm. Sec.*, 2008.

[14] F. Gauthier, T. Lavoie, and E. Merlo. Uncovering Access Control Weaknesses and Flaws with Security-discordant Software Clones. In *Annual Computer Security Applications Conference (ACSAC)*, 2013.

[15] S. Haefliger, G. von Krogh, and S. Spaeth. Code Reuse in Open Source Software. *Journal of Management Science*, Jan. 2008.

[16] IDA Pro - Interactive Disassembler. http://www.hex-rays.com/idapro/.

[17] J. Jang, A. Agrawal, and D. Brumley. ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In *IEEE Symposium on Security and Privacy*, 2012.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *International Conference on Software Engineering (ICSE)*, 2007.

[19] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary Function Clustering Using Semantic Hashes. In *ICMLA*, 2012.

[20] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary function clustering using semantic hashes. In *ICMLA*, 2012.

[21] R. Johnson and D. Wagner. Finding User/Kernel Pointer Bugs with Type Inference. In *USENIX Security Symposium*, 2004.

[22] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Softw. Eng.*, 2002.

[23] kingcope. Mikrotik RouterOS 5.* and 6.* sshd remote preauth heap corruption, September 2013. http://tinyurl.com/mikrotik-vuln.

[24] A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast Location of Similar Code Fragments Using Semantic 'Juice'. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, 2013.

[25] V. B. Livshits and M. S. Lam. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium*, 2005.

[26] I. McCabe Software. More Complex Equals Less Secure: Miss a Test Path and You Could Get Hacked, Feb 2012. http://www.mccabe.com/pdf/More%20Complex%20Equals%20Less%20Secure-McCabe.pdf.

[27] Microsoft-Research. Z3: Theorem Prover, February 2014. http://z3.codeplex.com/.

[28] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Commununications of ACM*, 1990.

[29] J. Ming, M. Pan, and D. Gao. iBinHunt: Binary Hunting with Inter-procedural Control Flow. In *International Conference on Information Security and Cryptology*, 2013.

[30] G. Myles and C. Collberg. K-gram Based Software Birthmarks. In *ACM Symposium On Applied Computing (SAC)*, 2005.

[31] NETGEAR. ReadyNAS OS Version 6.1.6, June 2014. http://tinyurl.com/ng-readynas-616.

[32] B. H. Ng and A. Prakash. Expose: Discovering Potential Binary Code Re-use. In *IEEE COMPSAC*, 2013.

[33] J. Pewny, F. Schuster, C. Rossow, and T. Holz. Leveraging Semantic Signatures for Bug Search in Binary Programs. In *Annual Computer Security Applications Conference (ACSAC)*, 2014.

[34] R. Seggelmann, M. Tuexen, and M. Williams. RFC 6520: Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension, Feb. 2012.

[35] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format-String Vulnerabilities with Type Qualifiers. In *USENIX Security Symposium*, 2001.

[36] Y. Shoshitaishvili. Python bindings for Valgrind's VEX IR, February 2014. https://github.com/zardus/pyvex.

[37] UBM Tech. Embedded Market Study, Mar. 2013. http://tinyurl.com/embmarketstudy13.

[38] US-CERT/NIST. Vulnerability Summary for CVE-2013-1813, November 2013. http://tinyurl.com/cve20131813.

[39] Valgrind. Documentation, 2014. http://www.valgrind.org/docs/.

[40] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory Errors: The Past, the Present, and the Future. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2012.

[41] E. Vanderbeken. How Sercomm saved my Easter!, April 2014. http://tinyurl.com/sercomm-backdoor.

[42] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[43] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *Symposium on Network and Distributed System Security (NDSS)*, 2014.

[44] R. B. Zadeh and A. Goel. Dimension Independent Similarity Computation. *CoRR*, abs/1206.2082, 2012.